**Programação(planejamento) dinâmica** = recursão com o apoio de uma tabela

-Estratégia de projeto de algoritmos

* Algoritmo recursivo = cada [instância](https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/instance.html#instance) do problema é resolvida a partir da solução de instâncias menores, ou melhor, de subinstâncias da instância original.
* A característica distintiva da programação dinâmica é a tabela que armazena as soluções das várias subinstâncias.  O consumo de tempo do algoritmo é, em geral, proporcional ao tamanho da tabela.
* Tabela: armazena as soluções das subinstâncias.
* O algoritmo recursivo é tipicamente ineficiente porque refaz, muitas vezes, a solução de cada subinstância.  Uma vez escrito o algoritmo recursivo, entretanto, é fácil construir uma tabela para armazenar as solução das subinstância e assim evitar que elas sejam recalculadas.  A tabela é a base de um algoritmo de programação dinâmica.

Programação dinâmica = recursão + tabelaSoluçãoCadaRecursão

Aplicação:

* É preciso que o problema tenha estrutura recursiva:  a solução de toda instância do problema deve conter soluções de subinstâncias da instância.
  + Ex.:  representada por uma [recorrência](https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/recorrencias.html) e a recorrência pode ser traduzida em um algoritmo recursivo.
* Uma recorrência é uma expressão que dá o valor de uma função em termos dos valores anteriores da mesma função.  Por exemplo,
* F(n)  =  F(n−1) + 3n + 2
* Para analisar o consumo de tempo de um algoritmo recursivo é necessário resolver uma recorrência.

## Exemplos

Eis alguns algoritmos que usam programação dinâmica:

* [algoritmo da subsequência crescente máxima](https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/sscm.html#dynprog2)
* [algoritmo da subsequência comum máxima](https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/sscm.html#LCS)
* multiplicação de cadeias de matrizes
* [algoritmo para o problema subset-sum](https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/mochila-subsetsum.html#dynamic-programming)
* [algoritmo para o problema da mochila booleana](https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/mochila-bool.html#prog-din)
* [algoritmo de Dijkstra](https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/dijkstra.html)

### Problema da Subsequência crescente máxima - Longest Increasing Sequence(LIS)

Esta página resolve o problema da subsequência crescente máxima.  Esse problema é uma boa introdução à técnica de [programação dinâmica](https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/dynamic-programming.html).

## Algoritmo recursivo:

Como no algoritmo recursivo, cada instância do problema é resolvida a partir da solução de suas subinstâncias.

**Subproblemas sobrepostos:**  
Considerando a implementação acima, a seguir é apresentada a árvore de recursão para uma matriz de tamanho 4. lis (n) fornece o comprimento do LIS para arr [].

lis (4)

/ |

lis (3) lis (2) lis (1)

/ /

lis (2) lis (1) lis (1)

/

lis (1)

Podemos ver que existem muitos subproblemas que são resolvidos repetidamente. Portanto, esse problema tem a propriedade Overlapping Substructure e a recomputação dos mesmos subproblemas pode ser evitada usando *Memoization* ou *Tabulation*. A seguir, é apresentada uma implementação tabulada para o problema do LIS.

## Algoritmo de programação dinâmica

* **LIS Longest Increasing Subsequence Size O(n2)**

/\* Dynamic Programming implementation of LIS problem

<https://www.youtube.com/watch?time_continue=393&v=Ns4LCeeOFS4> \*/

function LongestIncreasingSequence(arr) {

    if(arr.length ===0){ //verificacao

        return 0

    }

    let arrLIS = new Array(arr.length).fill(1)

    let max

    /\*Computar valores LIST otimizados de maneira ascendente\*/

    for(let i=1; i<arr.length; i++){

      for(let j=0; j<i; j++){

          if(  (arr[i] > arr[j]) ) { //temos uma LIS

                if( (arrLIS[j]+1) > arrLIS[i] ){

                    arrLIS[i] = arrLIS[j]+1

                    max = arrLIS[i]

                }

          }

      }

    }

    return(max)

  }

  /\* Complexity Analysis

  Time complexity : O(n^2). Two loops of n are there.

  Space complexity : O(n). arrLIS array of size n is used.

 \*/

 console.log( LongestIncreasingSequence( [  9, 9, 4, 2  ]))

* **LIS Longest Increasing Subsequence Size (N log N)**

<https://www.geeksforgeeks.org/longest-monotonically-increasing-subsequence-size-n-log-n/>

#### **Dynamic Programming with Binary Search:** <https://leetcode.com/problems/longest-increasing-subsequence/solution/>